國立交通大學

網路工程研究所

碩 士 論 文

軟串列：一種 NOR 快閃記憶體的原生索引結構

Soft List : A Native Index Structure for Nor-flash-Based Embedded Devices

研 究 生：許辰暉

指導教授：張立平　教授

中 華 民 國　九 十 七　年 七 月

軟串列:一種 NOR 快閃記憶體的原生索引結構
Soft List: A Native Index Structure for Nor-Flash-Based Embedded Devices

研 究 生：許辰暉　　Student：Chen-Hui Hsu

指導教授：張立平　　Advisor：Li-Pin Chang

國 立 交 通 大 學

網 路 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Network Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年七月

軟串列：一種 NOR 快閃記憶體的原生索引結構

學生：許辰暉　　指導教授：張立平

國立交通大學網路工程研究所碩士班

摘　　要

在嵌入式系統中，NOR 快閃記憶體主要用來儲存二進制的可執行碼。因為實際情況或價格的限制，許多裝置依然使用 NOR 快閃記憶體來儲存動態資料。一個在 NOR 快閃記憶體上的有效索引結構不僅減少中央處理器週期也延長裝置的使用壽命。然而，現有的索引結構因為 NOR 快閃記憶體的物理限制是很難應用在 NOR 快閃記憶體上。所以我們提出軟串列，一種 NOR 快閃記憶體的原生索引結構。軟串列透過實體指標組織資料，所以不需要位置轉換和開機掃描。基本的想法是一個指標能夠指到多個資料。該機制在搜尋上提供了快速跳躍的機會。當資料量大的時候，軟串列能夠擴增成多層的架構。軟串列最吸引人的是他簡單的資料結構，且在實驗中證實他的效率。

關鍵字：快閃記憶體（Flash memory），儲存系統（storage systems），嵌入式系統（embedded systems），索引結構（index structure）。

Soft List : A Native Index Structure for Nor-flash-Based Embedded Devices

Student：Chen-Hui Hsu Advisor：Dr. Li-Pin Chang

Institute of Network Engineering

College of Computer Science

National Chiao Tung University

Abstract

In embedded devices, NOR flash primarily serves as storage for binary executables. Due to limitations on form factor or cost, many devices also consider NOR flash as storage of dynamic data. The significance of efficient indexing over NOR flash are not only reduced CPU cycles but also prolonged operating periods. However, existing index structures are hardly applicable because of the physical constraints of NOR flash. Soft lists, a native index structure for NOR flash, are proposed. Soft lists organize data in terms of pointers of physical addresses, so address translation and initialization scan are not required. The basic idea is to allow a number of probes for de-referencing a data pointer. Interestingly, the probes provide opportunities for fast forward skips on search. Soft lists are then extended to be multilevel for scalability. The most attractive property of soft lists is its simplicity, and its efficiency has been verified by our experiments.

Keywords：Flash memory, storage systems, embedded systems, index structure

誌　　謝

終於抵達寫致謝詞的這一刻，這意味著研究所生涯即將正式的畫上句點，回顧兩年來的經歷，內心充滿幸福與感謝，首先誠摯的感謝指導教授張立平老師，老師悉心的教導使我得以了解嵌入式系統的深奧，不時的討論並指點我正確的方向，使我在這些年中獲益匪淺。因為有你的體諒及幫忙，使得本論文能夠更完整而嚴謹。

二年的求學生涯裡，讓我學習了不少新的知識和待人接物的方法，非常感謝，每個老師的敦敦教悔，而且勤而不懈的教導我，讓我能夠一路走來都很順利。

再來，感謝在這段期間，黃千庭、林松德、鄭家明、許蕙茹同學的幫忙，使得我能順利走過這兩年。實驗室的黃士庭蘇宥全、郭郡杰、楊明毅學弟、廖秀芬學妹們當然也不能忘記，你/妳們的幫忙及搞笑我銘記在心。

研究口試期間，感謝楊家玲老師、謝仁偉老師、陳雅淑老師不辭辛勞細心審閱，不僅給予我指導，並且提供寶貴的建議，使我的論文內容可以更臻完善， 在此由衷的感謝。

感謝系上諸位老師在各學科領域的熱心指導，讓我增進各項知識範疇， 在此一併致上最高謝意。

最後，謹以此文獻給我摯愛的雙親。

Contents

List of Tables

List of Figures

1 Introduction

Flash memory offers non-volatile storage at very approachable price. It is now an essential component in embedded devices. Currently two types of flash memory are widely used, namely NOR flash and NAND flash. NOR flash can be mapped to the processor's address space, so it is mainly for storing executable images. NAND flash offers very high information density with page-oriented operations, and it is a good choice for mass storage. However, due to considerations of form factor or cost, it may not be affordable to support both in a tiny embedded device. Many deeply embedded devices, such as network routers, solely use a piece of NOR flash for storage of executable images and dynamic data. In such cases, the NOR flash is divided into a readonly partition for executables and a read-write partition for dynamic data.

Efficient access of dynamic data has its significance to embedded devices, because both CPU cycles and energy are very precious resources. Various index structures have been developed for indexing data over byte-addressable RAM and block-oriented storage [2]. However, because of the physical constraints of NOR flash, they are not directly applicable. On NOR flash, a piece of data can not be overwritten unless it is erased. Erasure on NOR flash is carried out in terms of a block, which is typically 128 KB [3]. To avoid erasing a large block on every update, data are updated out of place. It invalidates all the pointers referring to the updated data. Invalid pointers can be revised by out-place updates, but it may in turn invalidate many other pointers. Even worse, activities for free-space reclaiming (i.e., garbage collection) involve data movements, but data movements themselves require extra free space for revising involved pointers. In the worst case, the system can be deadlocked.

Realizing efficient index structures over flash memory is very challenging, as illustrated above. The problem is addressed in past work [6, 5] by using logical addresses. In the rest of this paper, let a data object (sometimes simply an object) refers to a piece of data managed by index structures. Let a logical pointer be a pointer referring to logical addresses, and a physical pointer is defined accordingly. Every data object is assigned to a logical address, and logical pointers are used instead of physical pointers. With logical pointers, even if a data object changes its physical residence, its logical address remains unchanged. So data-objet updates are de-coupled from pointer updates. However, this approach has two major drawbacks. First, the logical-to-physical mapping information must be kept in RAM (which is capable of in-place updates) to avoid the chicken-and-egg problem. Such a RAM-space requirement is no doubt a burden for embedded devices. Second, the mapping information is unknown unless the entire flash memory is scanned. Even worse, it is in volatile memory. It contradicts that most embedded devices are required to be instantly operational after power is on.

Soft lists, a native index structure over NOR flash, is proposed. The major difference from existing approaches is the use of *soft pointers*. Soft lists are basically linear ordered lists, in which data objects are organized in terms of soft pointers. Unlike an ordinary physical pointer, a soft pointer simultaneously refers to many physical addresses. The key idea of using soft pointers is to allow a number of probes when de-referencing a soft pointer. The meanings are twofold: First, it is possible to move data objects around without invalidating any soft pointers. It greatly simplified the procedure of free-space reclaiming.

Second, on search of keys, it is possible to skip over a large amount of data objects if a random object is probed. It is referred to as *random forward skips* in the rest of this paper. Random forward skips are taken by just following any valid object found at the first probe. Interestingly, in most of the cases, it is even not important to know which object is intended to be referred to by a soft pointer.

Because soft pointers are based on physical addresses, the needs for address translation and initialization scan are completely eliminated. Although search with a soft list is surprisingly fast, after all it emulates a linearly ordered list. It is not guaranteed to scale well as the total number of keys is very large. To deal with this problem, a soft list is extended to be multilevel. A multilevel soft list is of a number of parallel independent soft lists, very similar to a skip list [10]. In a multilevel soft list, a data object has multiple soft pointers, one for each different level. Every data object hooks on the lowest-level soft list. The higher the level a soft list is, the lower the probability a data object hooks on it. Note that random forward skips are still taken for soft lists at any level. On search, high-level soft lists provide long-distance skips, and low-level soft lists are visited when the searched key is being closely approached. A multilevel soft list provides very good scalability, and its implementation is extremely simple. Even better, it is very friendly to flash memory because no expensive writes are required for self-reorganizing. We have conduct a series of experiments and comparison, for which we found that soft lists are much faster than linearly ordered lists and even skip lists.

The rest of this paper is organized as follows: Section 2 summarizes prior work related to implementing index structures over flash memory. Section 3 presents the design and implementation of soft lists. Section 4 includes our experimental results, and Section 5 concludes this paper.

2 Motivation

2.1 Characteristics of NOR Flash

Geometry		Timing	
Word size	2 Bytes	Word read	110 ns
Capacity	32M words	Word write	80 us
Block size	64K words	Block erase	0.6 s
Block endurance	100K cycles		

Table 1: The specification of a typical NOR flash [3].

NOR flash memory, as a kind of non-volatile memory, is write-once and bulk-erase. Initially NOR flash is entirely of free space. Data on NOR flash are byte-addressable, and a byte can be read for infinite times. NOR flash can be repeatedly written, provided that each write goes to different byte. However, successive writes to the same byte must be interleaved by block erasure. A NOR-flash block is typically 128 KB [3]. To avoid erasing a block on every update, data are updated out of place. The old versions of the updated data are considered as invalid. As writes keep arriving, the amount of free space on NOR flash would become low. Space occupied by invalid data is reclaimed by means of block erasure. Before a block is erased, all valid data on the block must be moved away. Activities for space reclaiming are referred to as *garbage collection*.

Each NOR-flash block individually tolerate a number of erasure operations. The limit is typically 100 K cycles under the current technology [3]. Any block exceeding the limit may suffer from unreliable data access, so it is desirable to evenly erase every block to postpone the appearance of the first worn-out block. *Wear leveling* is to manipulate data placement so that erasure can be directed to infrequently erased blocks [12, 13, 14]. The specification of a typical NOR flash is shown in Table 1.

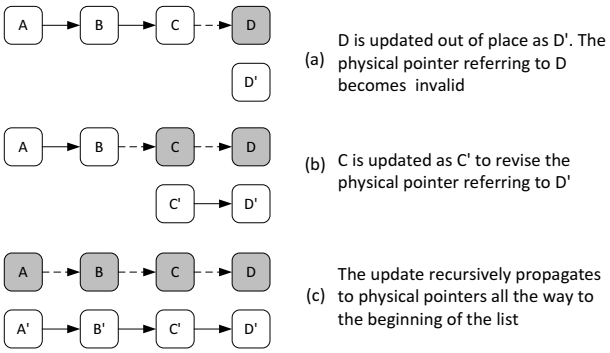

Figure 1: Propagation of Physical-Pointer Updates.

2.2 Physical Pointers vs. Logical Pointers

This section addresses the issues of using physical pointers over flash memory, as well that of using logical pointers.

For RAM, it is taken for granted to refer to data in terms of physical addresses, because RAM is capable of in-place updates. For flash memory, each update to a piece of data may invalidate all the physical pointers referring to it, because new data must be written to a new physical location on update. Consider the example shown in Figure 1: A list of four data items are organized with physical pointers. On the update of D, the new data D' are written to some available space on flash memory. However, it is not possible to rewrite the physical pointer of C to correctly refer to D', so C must also be updated out of place. As a result, physical pointers (data objects as well) all the way to the beginning of the list must in turn be updated. This problem is referred to as *pointer-update propagation*.

Flash-memory management activities is also a cause of moving data objects. Before a block is erased for garbage collection, all valid data must be moved away from the block. The data movements invalidate all physical pointers referring to the moved data, and all the physical pointers must be updated out of place. Activities for reclaiming free space as such, may introduce the needs for more free space. Potentially the system can be deadlocked. This problem is referred to as *garbage-collection deadlocks*.

To deal with the problems of using physical pointers over flash memory, one can choose to use logical pointers. In this approach, each piece of data is assigned to a logical address. Data are referred to by logical addresses instead of physical locations. To implement logical pointers, a translation table must be maintained in RAM, which is capable of in-place updates. The table is indexed by logical addresses, and each table entry is of a physical address. On out-place updates of a data object, its new physical address is revised in the table. So data-object updates are completely detached from pointer updates.

Even though the use of logical pointers eliminates the problems caused by physical pointers, they introduce scalability issues. Specifically, the RAM-resident translation table is a burden for small embedded devices in terms of not only hardware cost but also energy budget. Even worse, to bind all the logical addresses to physical addresses, it is necessary to scan the entire flash memory. The scanning procedure imposes lengthy delay on system initialization, which is intolerable in many embedded devices. The scalability problem is exaggerated especially when the flash memory is large.

2.3 Related Work

There have been many excellent index structures developed for byte-addressable RAM or block-based storage [2]. However, they are not directly applicable to flash memory. Prior work on indexing over flash memory are mainly based on the use of logical addresses. Wu et al. [6] and Xiang et al. [7] propose to implement B-tree over flash memory, and a node translation table is used to map B-tree nodes to flash-memory pages. Lin et al. [5] propose a new design of hash tables for flash memory, for which a bucket is mapped to a collection of related flash-memory pages. Actually, maintaining the mapping of logical addresses to physical addresses is not an issue specific to index structures. For

flash-based disk emulation [1, 11, 16], the mapping is from disk-sector numbers to flash-memory locations. A native flash file system [17, 18] should map an i-node number with a byte offset to flash-memory addresses.

The use of logical addresses introduces two technical issues. The first problem is that a RAM-resident translation table is required. The translation table costs precious RAM space and energy. For reducing the RAM-space requirements, Chang et al. [11] developed a variable-granularity scheme for address translation. Kim et al. [15] and Lee et al. [16] propose to reduce the table size by adopting block-level translation instead of page-level translation. As to energy consumption, although it is not formally addressed in past work, its significance can be verified by the following observations: A typical SDRAM [4] needs 70 mA in operating mode, 20 mA in standby mode, and 160 mA for refresh every other 64 milli-seconds. A typical NOR flash [3] needs 30 mA in operating mode, 30 mA for read/write/erase, and 10 μA in standby mode.

Besides the costs of space and energy, the other problem of using logical addresses is that the mapping is unknown until the entire flash memory is scanned. Wu et al. [8] propose to incrementally commit summary information onto flash memory to help to speed up the scanning procedure. Yim et al. [9] propose to compress the entire mapping information and put it in some handy locations.

Regardless which techniques are taken for reducing RAM footprint and for shortening scanning time, these overheads are inevitable. Different from prior work, this work considers organizing data in terms of physical addresses. We aim at completely removing the needs for address translation and initialization scan.

3 Design and Implementation of Soft Lists

This section proposes soft lists. We shall first present simple soft lists, which emulate linearly ordered lists. Simple soft lists are then extended to multilevel soft lists for better scalability.

3.1 Simple Soft Lists

3.1.1 Index Objects and Soft Pointers

An index object is the smallest unit for indexing with soft lists. It is of a key, a value, and a soft pointer referring to another index object. Basically the length of the keys can be arbitrary, so there can be infinitely many other keys between any two keys. For simplicity, we use fixed-length slots for storing keys. A value is as large as a key in size. We assume that each update to the value rewrites the entire index object. Note that the value can also be a soft pointer referring to a data page. For the ease of presentation, index objects, objects, and keys are interchangeably used to refer to the same thing in the rest of this paper.

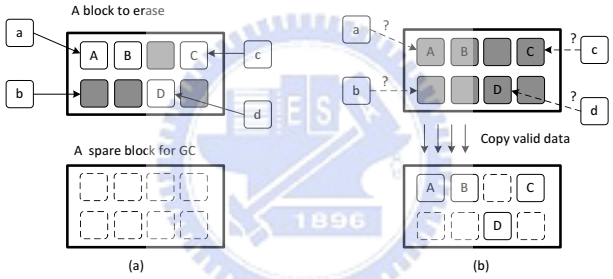

Figure 2: A problem caused by garbage collection. (a) A block has valid data A, B, C, and D. (b) Before the block can be erased, all valid data are copied to a spare block. Four data objects are forcibly moved, and all the physical pointers referring to them become invalid.

As mentioned in Section 2.2, the physical residence of index objects may involuntarily be changed because of the needs for garbage collection. Figure 2(a) shows an scenario, in which a block has valid objects A, B, C, and D. To erase the block for garbage collection, beforehand all the valid objects must be moved to a spare block. However it invalidates all the physical pointers previously referring to the objects, as shown in Figure 2(b). To revise the physical pointers, objects a, b, c, and d can be rewritten out of place. However, recursively another batch of object rewrites could be triggered. Because free space is consumed before free space can be reclaimed, the system might be deadlocked.

Soft pointers are proposed to approach the problem. The basic idea is to allow a number of "probes" when referring to an index object. In other words, if an index object is involuntarily moved, it is moved to some pre-determined locations. On access, a soft pointer may not immediately refer to the desired index object, but the correct one can always be found after some probes. The

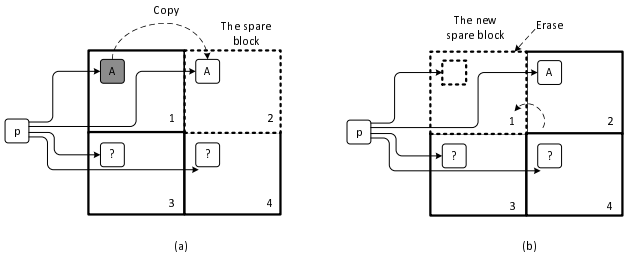

Figure 3: Index object "p" refers to index object "A" by means of a soft pointer. A turnstile is of four blocks including a spare block. (a) Object A is shifted to the spare block and then block 1 can be erased for garbage collection. (b) The soft pointer does not lose object A after erasing block 1, because it refers to all the index objects having the same block offset in a turnstile.

use of soft pointers removes pointer updates from garbage-collection activities, and therefore deadlocks are avoided.

Soft pointers are realized by means of *turnstiles*. A turnstile is a group of NOR-flash blocks, and the entire NOR flash is partitioned into turnstiles. Every turnstile is allocated to some spare block for garbage collection, and garbage collection is confined to each individual turnstile. Figure 3(a) shows a turnstile, in which there are four blocks and block 2 is a spare block. Consider that object "p" outside the turnstile refers to object "A" inside the turnstile by means of a soft pointer. It is called an *inter-turnstile reference*. Now suppose that block 1 is chosen for erasure. Object A is "shifted" to block 2, and block 1 is then erased into a new spare block, as shown in Figure 3(b). Logically the turnstile is "rotated" counterclockwise for garbage collection. A valid object is always shifted to a spare block with the same block offset, and a soft pointer refers to all the index objects having the same block offset. As turnstile rotates, the soft pointer never loses object A, if up to four probes are allowed. Other than object A, the referred objects in blocks 3 and 4 are random objects.

If an object refers to another object in the same turnstile, then it is called an *intra-turnstile reference*. For intra-turnstile reference, a soft pointer is the relative distance between an object to the referred object. In this case, the maximum number of probes needed is the number of spare blocks in a turnstile. Because the worst case is that all the spare blocks are in-between the two objects. For example, in Figure 3, up to 1 extra probe is needed. Besides that fewer probes are needed by intra-turnstile references, the rationale to rotate a turnstile on garbage collection is to consider wear leveling. As a turnstile rotates, every block is in turn erased so wear leveling is perfect. However, readers may notice that if the block to erase is far from the spare block(s), then the overheads of object copy and block erasure is high. We shall again address this issue in Section 3.3.

For the ease of presentation, some terms are defined here: Let the *degree* of soft pointers be how many blocks a turnstile has. Including the intended object, a soft pointer refers to many objects. Let the intended object be the *target object* (e.g., object A in Figure 3), and all the others be the *buddy objects* of the soft pointers.

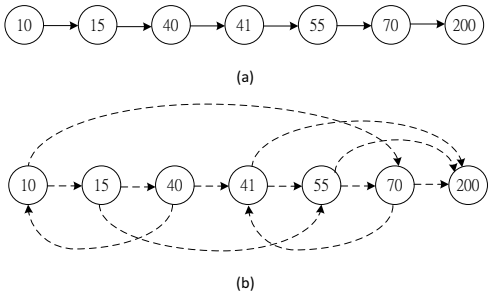

Figure 4: (a) A linearly ordered list. (b) A soft list with soft pointers. The degree of the soft pointers is two.

3.1.2 Search with Simple Soft Lists

This section introduces simple soft lists, which organize index objects with soft pointers. We are particularly interested in how search is carried out with soft lists and how soft pointers benefit search.

Let us first be focused on search with a linearly ordered list. A linearly ordered list is based on physical pointers. With the example shown in Figure 4(a), to locate a key, starting from the first index object, iteratively we move forward until the current key is no smaller than the key to find. The desired key is found if the current key equals to it, otherwise it does not exists. So to locate key 200, we need to visit 7 objects.

A soft list, which emulates a linearly ordered list, organizes index objects with soft pointers. Figure 4(b) shows an example on soft lists, in which the degree of soft pointers is two. Different from search with a linearly ordered list, we move forward until the keys of *all* the probed objects are no smaller than the key to find. Note that, on de-referencing a soft pointer for search, it is not required to distinguish its target object from the buddy objects.

For example, in Figure 4(b), suppose that we are to find key 200. In the beginning we start from key 10, and the next keys may be 70 and then 200. Surprisingly, it takes only three steps. On search, it is possible that we skip too far and must fall back to try again. For example, to search key 55, starting from 10, for the first trial we go to 70, which is larger than 55, so we fall back to 10 and try again. For the second trial we get 15, which is smaller than 55, and the search continues. From key 15, we go straight to 55 and report the key is found. The search takes three steps only. Another case on search is that the desired key does not exist. If we are to search 16 for example, from key 10, key 15 is visited. From key 15, for the first trial we have key 40, which is larger than 16. For the second trial, we have key 55, which is also larger than 16. After trying all the possibilities, it is reported that key 16 can not be found.

Because buddy objects are random objects, on search with soft lists it is possible to skip over a large amount of index objects. However, by this way backward moves are also possible. For example, suppose that we are to search key 41 and currently we are at key 40. From key 40 it is possible to move backward to key 10. In this case, we shall fall back to key 40 and try again, and this time we shall find key 41. Backward moves are of course overheads of using soft lists, and we shall address this issue again in Section 3.2.2. Algorithm 1

Algorithm 1 Search with a simple soft list

```
Require: curr: the first (leftmost) index object,
    key: the key to locate
Define: curr→next: a soft pointer,
    curr→next[i]: the i-th probe of the soft pointer.
 1: while curr→key != key do
 2:   for each curr→next[i] do
 3:     if curr→next[i]→key ¡ curr→key then
 4:       continue; {avoid backward moves}
 5:     end if
 6:     if curr→next[i]→key ≤ key then
 7:       curr = curr→next[i];
 8:       break;
 9:     end if
10:   end for
11:   if curr→next[i]→key ¿ key then
12:     return NOT_FOUND; {all probes have been tried}
13:   end if
14: end while
15: return FOUND;
```

shows the procedure of searching with a simple soft list.

3.1.3 Insertion/Deletion and Spare Pointers

Different from search, insertion/deletion involve modifications to soft lists. Let us first be focused on insertion. Because soft lists emulate linearly ordered lists, a new object is always inserted right after the object which's key is immediately smaller than the new key. Let the object be the *immediate predecessor* of the new object. If the new object is always written as a buddy object of the immediate predecessor, then the predecessor's soft pointer needs not change. However, it is infeasible because the total number of buddy objects a soft pointer can have is limited. Instead, any free space is eligible for storing the new object. To refer to the newly inserted object, the predecessor's soft pointer must be revised.

As mentioned previously in Section 2.2, updates to pointers inevitably pose the propagation problem. To address this, for each index object, we propose to reserve some empty slots as *spare pointers*. In the rest of this paper, the terms spare pointer and spare slots are interchangeably used. Initially a soft pointer occupies the first spare slot. As the soft pointer is revised, updates are in turn logged in empty spare slots. Figure 5 shows that an index object has four spare pointers, and initially the first is a soft pointer referring to object A. Suppose that the soft pointer is to be revised to refer to a newly inserted object B. A new soft pointer is written to the second slot, and then that in the first slot becomes invalid. The third spare pointer is in turn used for referring to another object C. By this way, until all the spare pointers are used up, an index object may revise its soft pointer for many times. The idea of spare pointers is feasible thanks to that NOR flash is byte-addressable.

When an index object runs out all its spare pointers, to refresh its spare slots, we may choose to rotate the turnstile it belongs to until the object is shifted

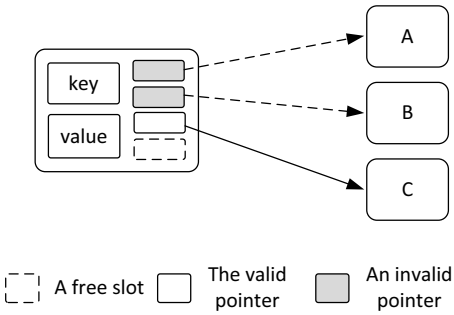

Figure 5: An index object, in which there are four spare slots. The object's soft pointer is in turn revised to refer to objects A, B, and C. The fourth slot is not yet used.

to a spare block. However, it might be too costly, if garbage collection is not necessary at that time. Instead, we propose to rewrite the index object out of place. As readers may notice, to rewrite an object may in turn triggers pointer updates. But the use of spare pointer can largely slow down the propagation of pointer updates. For example, if one object has eight spare pointers, then pointer updates won't be propagated to three other objects until an object is updated out of place for $8^3 = 512$ times. Whenever necessary, turnstiles can be rotated to completely stop any propagation. The use of spare pointers of course has its drawbacks. Specifically, the spare pointers must be scanned to identify which one is the valid pointer. However, in contrast to rewriting index objects, the cost of pointer scan is acceptable, because NOR flash reads much faster than writes, as shown in Table 1. We shall provide evaluation on this in our experiments.

With spare pointers, before a new object is inserted, its immediate predecessor must be located. To locate the location for insertion and the immediate predecessor, starting from the first object in a soft list, repeatedly we move forward to the next object via soft pointers until all the referred objects' key are larger than the key to be inserted. At this point, the current object is the immediate predecessor.

For example, to insert key 16 to the soft list in Figure 4(b), starting from key 10, the next key can be 200. Because 200 is larger than 16, we fall back to 10, and for the second probe we have 15, which is smaller than 16. So we move forward to 15, and then get the next key 40. Key 40 is larger than key 16, so we fall back and try again to de-reference the soft pointer of key 15, and this time we have 55. Because both 40 and 55 are larger than 16, we stop here and report that the immediate predecessor of 16 is 15. The procedure to find immediate predecessors is very similar to that in Algorithm 1, except that, when the algorithm reports "NOT_FOUND" (i.e., Step 12), "curr" is the immediate predecessor. A new object is then inserted right after the immediate predecessor, and then the inserted object refers to the predecessor's prior target object.

Deletion is carried out with a similar procedure. The immediate predecessor of the deleted object is first located, and then the predecessor's soft pointer is revised to refer to the target object of the deleted object.

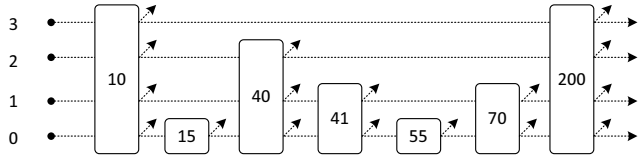

Figure 6: A multilevel soft list, which is of four simple soft list. Index objects hook on soft lists by means of soft pointers with degree=2. References for random forward skips are not drawn.

3.2 Multilevel Soft Lists

This section shows how simple soft lists are extended to multilevel soft lists. The design of multilevel soft lists is extremely simple, while they are expected to have good scalability when the total number of keys is large.

3.2.1 Structure of Multilevel Soft Lists

With random forward skips, data access over simple soft lists can be much faster than over linearly ordered lists. However its scalability is not guanranteed. One choice is to extend soft lists to balanced trees (such as red-black trees), since essentially these index structures are organized by lists. However, these trees require a large number of pointer updates for balancing activities, which in turn trigger many rewrites of index objects.

Instead of balanced trees, we propose to extend simple soft lists to multilevel soft lists. A multilevel soft list is composed by parallel independent simple soft lists, each of which is at different levels. Level 0 is the lowest level. An index object has $n+1$ soft pointers, one for each level. The connectivity between index objects and lists at different levels is controlled by a parameter $0 < p < 1$. Let q_1, q_2, ..., and q_n be n randomly generated numbers which are between 0 and 1. Let q_0=0. An index object hooks on the i-th level soft list only if $(q_0 \leq p) \wedge (q_1 \leq p) \wedge (q_2 \leq p) \wedge ... \wedge (q_i \leq p)$. In other words, the probability that an object hooks on the level-n soft list is p^n.

Figure 6 shows a four-level soft list, extended from the simple soft list in Figure 4. Readers may notice that a multilevel soft list is structurally similar to a skip list [10]. Each index object hooks on soft lists by means of soft pointers. Note that references for random forward skips of soft pointers are not drawn. Search with a multilevel soft list is very similar to search with a simple soft list, since conceptually we can treat all the soft pointers at different levels as one single soft pointer with a large degree.

Suppose that we are to locate 41. For the ease of discussion, let's first ignore buddy objects when de-referencing soft pointers. Starting from the highest level of key 10, the next key is 200, which is larger than 41. So we fall back to 10 and move downward to level two. This time we get 40, which is smaller than 55, so we move forward to it. From the second level of key 40, the next key is 200, so again we fall back and move downward to level one, and finally 41 is found.

Algorithm 1 can be slightly revised to support search with multilevel soft lists: One variable is needed to remember at which level currently we are, and search always starts from the highest level. Step 12 of Algorithm 1 is revised as moving downward to the next level. But if we are already at the bottom level,

then NOT_FOUND is reported. Note that, since now an object in multilevel soft lists has multiple soft pointers, the spare slots should be shared by all its soft pointers. There are simple and effective methods for reducing the overheads of scanning spare pointers, which are omitted here.

Different from search, insertion/deletion involve soft-pointer updates. For example, consider that we are to delete key 40 from the multilevel soft list in Figure 6. Key 10's soft pointers at level 2 and level 1 should be revised to refer to key 200 and 41, respectively. Key 15's soft pointer at level 0 should be revised to refer to key 41. In other words, at every level, the immediate predecessor of the deleted object inherits the deleted object's soft pointer. For the algorithm revised for search with multilevel soft lists, right before we move downward, the current object (i.e., "curr" in Algorithm 6) is an immediate predecessor of the object to be deleted. The total number of soft pointers to revise equals to the height of the deleted object.

3.2.2 Analysis on Multilevel Soft Lists

In a multilevel soft list, a high-level soft list intends to skip with far distances, because the total number of index object hooking on a high-level soft list is expected to be small. For example, in Figure 6 the second-level soft pointer of key 40 straight goes to the last key. However, with soft lists, it is not guaranteed that we can skip that far, because a soft pointer is sometimes de-referenced to buddy objects.

Consider a multilevel soft list with total N index objects and the probability parameter p. Suppose that we are to de-reference a level-i soft pointer. The expected number of index objects hooking on the level-i soft list is $N \cdot p^i$ objects, so in average the skip distance between the index object and its target object is

$$N/(N \cdot p^i) = p^{-i}$$

. Now consider that a soft pointer which's degree is five. Besides its target object, the four buddy objects are expected to be randomly distributed. When de-referencing the soft pointer for forward skip, there might be several cases, as shown in 7: The first is a backward skip. In this case, we fall back and try to de-reference the soft pointer again. The second case is a short skip, for which we move to a buddy object which has a key smaller than the target object's key. For third case, we happen to move to the target object. For the fourth case, we move to a buddy object, which has a key larger than the target object's key.

It is an important question how frequent short skips would be experienced. Not only short skips slow down search, but also it's costly to identify short skips. Note that simple soft lists never suffer from short skips because there are no other objects between an object and its target object. Consider the gray section in Figure 7, which is the expected distance between an level-i object and its target object, i.e., p^{-i}. Let the degree of a soft pointer be $d+1$. Since buddy objects are randomly distributed, the expected distance between two adjacent buddy objects is N/d objects. The probability of having short skips is

$$\frac{p^{-i}}{N/d}$$

. Now consider a multilevel soft list with n levels, and we are to search the key of the rightmost object. Let N_i be the total distance traveled at the level-i soft

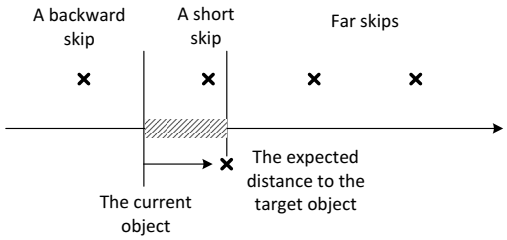

Figure 7: Possible results of de-referencing a high-level soft pointer with degree=5. The gray section stands for the distance of skipping to the target object. The keys of the four buddy objects are randomly distributed.

list. Before moving downward to the level-$(i-1)$ soft list, the expected number of objects visited at level i is $N_i p^i$. Therefore, the expected number of short skips at all levels is

$$\sum_{i=0}^{n-1} \left(\frac{p^{-i}}{N/d} \cdot N_i p^i \right)$$

. Because $\sum_{i=0}^{n-1} N_i = N$, it becomes

$$\frac{1}{N/d} \cdot N = d$$

. Interestingly, it is independent of N, i, and p. If we take $d = 5$, then no matter how many objects are there, the expected total number of short skips remains 4. Furthermore, as short skips are compensated by long skips, the net effect of short skips could be negligible, especially when the number of objects is large. We shall provide evaluation on this in our experiments.

3.3 Space Allocation and Wear Leveling

Index objects should be randomly distributed over the entire NOR flash because the buddy objects of a soft pointer are expected to be random objects. As readers may notice while reading on, the design of multilevel soft lists highly relies on randomness. The rationales behind are 1) to benefit search by random forward skips and 2) to properly estimate the costs of short skips.

For this purpose, we choose to allocate free space from a randomly selected block. Free space is allocated to an index object when 1) the object is newly inserted, 2) the object is rewritten to change its value, and 3) the object is rewritten to refresh all its spare pointers. As free space in blocks keep being consumed, block erasure is then needed to reclaim space occupied by invalid objects. Suppose that a block has been chosen as a victim for erasure (how it is chosen is discussed later). Before the victim block is erased, the turnstile it belongs to is rotated until all the valid objects in the block are shifted to a spare block, as mentioned in Section 3.1.1. However, as readers may notice, if the spare block is far behind the victim block, then all the blocks in-between are involved, and a lot of block erasure and valid-object copy are needed.

Alternatively, we can choose to directly shift all the valid objects from the victim block to the spare block. The victim block is then erased. The benefit of

this approach is that it does not involve the blocks between the victim block and the spare block. The drawback of this approach is that the maximum number of probes needed by intra-turnstile reference and inter-turnstile reference to locate a target object become the same. However, as mentioned in 3.1.2, most of the time, it is not even necessary to tell the target object from buddy objects. Besides, NOR flash reads much faster than writes, so the extra reads can be compensated by reduced data write and block erasure. Comparing to rotating turnstiles, wear leveling with this method needs more discussions.

Wear leveling is to evenly erase all the blocks so that the appearance of the first bad block is postponed as late as possible. The cause of uneven lifetime of blocks is closely related to free-space allocation. If frequently updated objects are clustered together in blocks, blocks will not be evenly erased because garbage collection favors blocks having many invalid objects. Our free-space allocation policy favors not only to distribute (buddy) objects over blocks randomly but also to direct block erasure to all the blocks evenly. It is to write new objects to free space allocated from randomly chosen blocks. By this way localities in access pattern are largely weakened, and invalid objects could be evenly distributed over blocks. Therefore, the overheads of erasing different blocks would be close. On space allocation, if a random block is chosen but there is no free space left, then the block becomes a victim block. Because the selection is completely random, even if a block is of a lot of immutable objects, eventually it will be chosen for erasure in favor of wear leveling.

4 Experimental Results

4.1 Experimental Setup

The proposed soft lists are evaluated by means of a simulator. The simulator considers the geometry of a real-life NOR flash [3]. Ignoring the boot blocks, in our experiments the NOR flash has 128 64K-word blocks. For each run of experiments, unless explicitly specified, the following basic settings are adopted: Each block is of 256 256-words slots for storing index objects. Besides soft pointers, an object has 6 extra spare slots for logging pointer updates. The 128 blocks are organized as 32 turnstiles, so each turnstile is of four block. One among the four is a spare block. Before each run of experiments, 12,000 different keys are sequentially inserted, and then the keys are randomly updated until each key is updated at least two times. It intends to trigger garbage collection so that every turnstile is rotated several times. Without this, the advantages of soft lists will be largely limited because many soft pointers would directly reach its target object with the first probe. On the completion of this setup procedure, all the experimental statistics (e.g., the accumulated numbers of reads and writes) are reset.

The proposed soft lists are evaluated against a scheme that is based on logical addresses, which is commonly taken in past work for organizing data over flash memory [6, 7, 17]. In this approach, each valid object is associated with a unique logical address (i.e., the key). A RAM-resident translation table is required for address translation. As an object changes its physical residence due to out-place update or garbage-collection activities, its logical address remains, so no pointer updates are required. By this way blocks need not to be organized as turnstiles. Note that pointer update is still needed when an object refers to another new object. To avoid rewriting an entire object on every pointer update, like an object in soft lists, each object reserves some spare slots for logging pointer updates. As logical pointers are not affected by out-place updates, this scheme is to serve as a trivial low bound of write traffic in our experiments.

The performance metrics are straightforward: the total number of word reads, word writes, and block erasure. Note that the numbers of reads and writes do not include those for garbage-collection activities. Garbage collection overheads are measured in terms of the total number of blocks erased. For wear leveling, it is indexed in terms of the distribution of erasure-cycle counts of all the blocks.

4.2 Simple Soft Lists

In this section, simple soft lists are evaluated against linearly ordered lists. A linearly ordered list is a singly-linked list ordered by keys, and it is based on logical pointers, as mentioned in the last section. In the rest of this paper, let **SSL** refer to a simple soft list, and **LOL** be a linearly ordered list.

4.2.1 Read-Only Queries

In the first part of experiments, we are concerned with the usefulness of random forward skips. It is evaluated by performing read-only queries. For each run of experiments, all the keys are queried with three different patterns. In the first

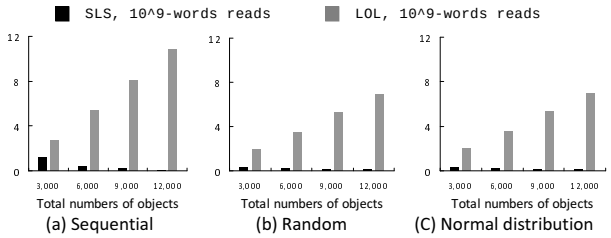

Figure 8: The total number of word reads of SSL and LOL with respect to different total numbers of keys. All the keys are queried with (a) a sequential pattern, (b) a random pattern, and (c) a normal distribution of query frequencies over keys.

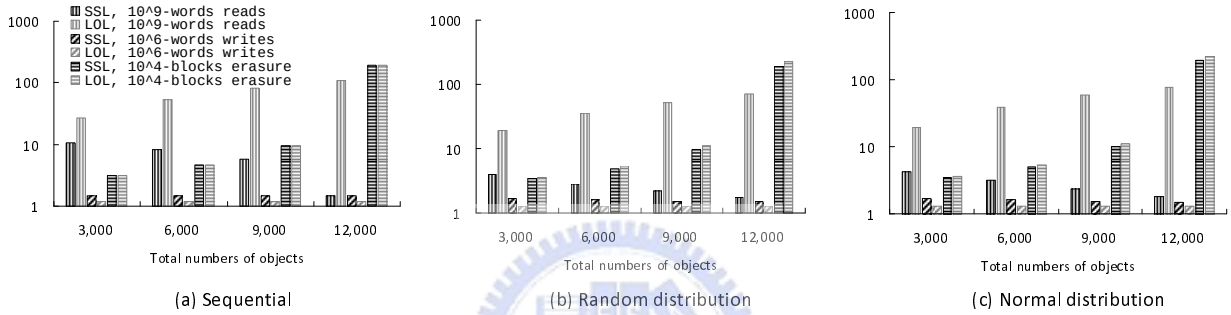

Figure 9: The total numbers of reads, writes, and block erasure of SSL and LOL with respect to different total numbers of keys and different access patterns. Note that the Y-axes are of logarithmic scales.

pattern, all the keys are sequentially queried. The second pattern randomly queries all the keys. The third access pattern is a normal distribution of query frequencies over keys. The mean of the distribution is the median of all the keys, and the variance is one-sixth the total number of the keys. The total number of keys varies from 2,000 to 16,000.

Figure 8 shows the total number of word reads with difference access patterns. Read overheads comes from traversing SSL or LOL for locating a key and scanning spare pointers to find the valid pointer. LOL's total number of reads increases linearly with the total number of keys, as it performs linear search. Because of random forward skips, the total numbers of reads of SSL are significantly smaller than that of LOL in all the cases. Interestingly, SSL's total number of reads gradually decrease as the total number of keys is large. The rationale is that, when the total number of keys is small, the possibility is high that a soft pointer's buddy object is an invalid object. If an invalid object is probed, probing is carried on until an valid object is found. Very likely that the target object is found as the first valid object, and in this case random forward skips are not taken. If the total number of (valid) objects is large, then random forward skips are taken with high frequencies. It can be verified by results in Table 2, which show the average skip distance of SSL and LOL. The skip distances of SSL are much longer than that of LOL.

Another observation on Figure 8 is that SSL's performance is not much

affected by different access patterns. It is a characteristic of using soft pointers. Since the key of buddy objects are random objects, the total numbers of objects visited for finding different keys are close. On the other hand, LOL is very slow on sequential access because it uses linear search.

total # of keys	3,000	6,000	9,000	12,000
LOL	1.0	1.0	1.0	1.0
SSL	6.0	16.3	29.5	42.9

Table 2: The average skip distances of LOL and SSL with respect to different total numbers of keys. The access pattern is a normal distribution.

4.2.2 Insertions and Deletions

This part of experiments examine SSL's performance with updates. Each run of experiments still uses the same setup procedure. After setup, all the keys are updated with the three access patterns. For updating a key, the key is first deleted and then re-inserted. It intends to introduce the needs for pointer updates.

Figure 9 shows the total numbers of reads, writes, and block erasure of SSL and LOL with respect to different total numbers of keys and different access patterns. Note that the Y-axes are of logarithmic scales. For reads, SSL still win its edge over LOL. Although it can not easily be seen in the figures, the advantage is not that significant as in read-only queries. That is because SSL spends extra reads to locate the immediate predecessor of a deleted/inserted object, as mentioned in Section 3.1.3. Nevertheless, SSL still reads a much smaller amount of words than LOL.

For writes, SSL writes slightly more words than LOL. Note that the overheads of writes do not include those for garbage collection. In average SSL requires 20% more word writes than LOL. That is because logical pointers of LOL are not affected by out-place updates. We must emphasize that, since logical addresses are used, LOL serves as a trivial low bound for the total number of writes. As to SSL, because objects are referred to by physical addresses, when objects are updated out of place, related soft pointers are revised by logging changes in spare slots. Necessary object rewrites to refresh spare slots are performed accordingly.

For garbage-collection overheads, it can be seen that SSL and LOL erase nearly the same number of blocks. It can also be noted that how many blocks erased by SSL is not much affected by using different access patterns. That is because free space is allocated from randomly selected blocks. As a result, invalid objects are evenly distributed over blocks, regardless the patterns of object updates. Also, both SSL and LOL erase a large number of blocks when the total number of keys is large. That is because erasing a block would involve a large number of valid objects, and the objects must be copied to before block erasure. The net amount of free space reclaimed every block erasure is relatively low, and many block erasure are needed to reclaim a specified amount of free space.

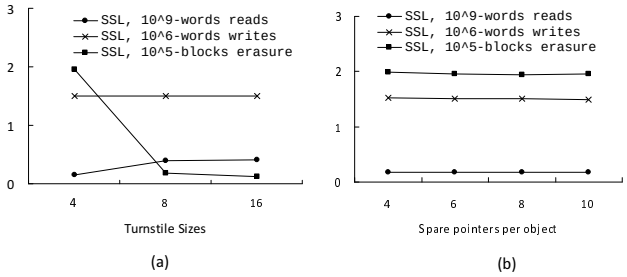

Figure 10: The total numbers of reads, writes, and block erasure of SSL with (a) different turnstile sizes and (b) different numbers of spare pointers in an object. The access pattern is update with a normal distribution.

4.2.3 Turnstile Sizes and Spare-Pointer Numbers

This part of experiments aim at evaluating different organizations of simple soft lists. The setup procedure for experiments is the same as in prior sections. The access pattern is to update keys with a normal distribution.

First we vary the turnstile size (i.e., the total number of blocks in each turnstile). As the results in Figure 10(a) show, the total number of reads increases with the turnstile size. That is because the finding of the predecessor of an object requires probing all the buddy objects. The larger the turnstiles are, the more number of probes are needed. Another source of the extra reads is invalid objects. Because each turnstile has one spare block, with large turnstiles the total number of non-spare blocks would be large. As a result, the total number of invalid objects is large too. In this case, when de-referencing a soft pointer, it is very possible that an invalid object is found as a buddy object. Since invalid object is of no use, extra reads are needed to probe until a valid objects is found.

As to writes (except those for garbage collection), they are irrelevant to turnstile sizes, so it can be seen that the total numbers of writes are all the same in experiments. For garbage collection, it is shown that, when turnstiles are small, the costs of block erasure become very high. As mentioned previously, the smaller the turnstiles are, the smaller the total numbers of non-spare blocks are. In this case, block erasure involves a large number of valid objects, and therefore the reclaiming of free space becomes very slow. It can be seen that the setting of the turnstile size is a trade off between read costs and garbage-collection overheads. As NOR flash erases much slower than it reads (see Table 1), one possible approach is to trade space for performance. That is, use larger NOR flash with small turnstiles.

Figure 10(b) shows the results of varying the total number of spare slots of an object. Note that the object size is not affected by the total number of spare slots in an object. Let us first consider the write costs. If there are many spare pointers in each object, then to rewrite an object to refresh all its spare pointers is not frequently needed. By this way then write overheads can be significantly reduced. However, as the results show, the total numbers of writes are not much affected by the numbers of spare pointers. It is an evidence on that an object is re-written before it runs out of spare pointers. In other words, the number of spare pointers in an object needs not be large.

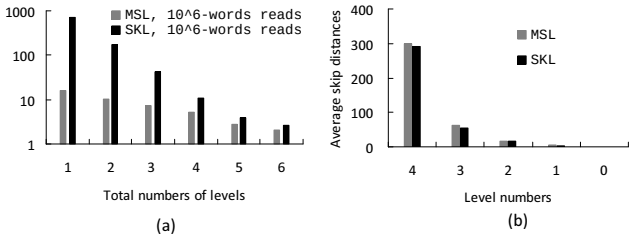

Figure 11: (a) The total number of words that MSL and SKL read with respect to different total numbers of levels. The access pattern is to query with a normal distribution. (b) The average skip distances at different levels of MSL and SKL. The total number of levels is five. The Y-axes are of logarithmic scales.

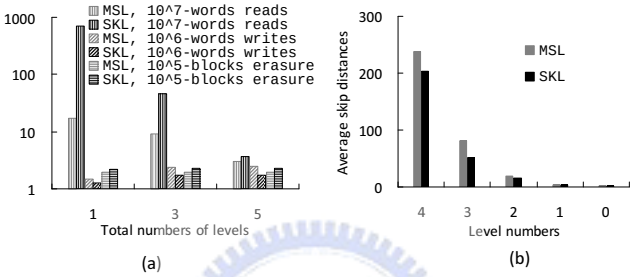

Figure 12: (a) The overheads of reads, writes, and erasure of MSL and SKL with respect to different total numbers of levels. The access pattern is to update with a normal distribution. (b) The average skip distances at different levels of MSL and SKL. The total number of levels is five. The Y-axes are of logarithmic scales.

4.3 Multilevel Soft Lists

In this section, multilevel soft lists are evaluated and compared against skip lists [10]. We are particularly interested in whether multilevel soft lists provide good scalability as skip lists do. In our experiments, skip lists are implemented based on pointers of logical addresses, as is LOL. In the rest of this paper, let multilevel soft lists be denoted as **MSL**, and **SKL** refer to skip lists.

We shall first confine our attention to read-only queries. The same setup procedure for SSL/LOL is used. The total number of keys inserted is 12,000, and the access pattern is a normal distribution of query frequencies over keys. The probability parameter p is 0.25, as suggested in [10]. The total number of levels that MSL and SKL have vary from 1 to 6. Figure 11(a) shows the read overheads of MSL and SKL with respect to different total numbers of levels. MSL uses much fewer word reads than SKL when the total number of level is small. As the total number of levels increases, as expected, the read overheads of SKL dramatically decrease. The read overheads of MSL also drop exponentially as the total number of levels increases. But MSL's read overheads do not drop as fast as that of SKL, because MSL's read overheads are already very low when there is only one level. For MSL, let the average skip distance be defined as the average number of objects in MSL skipped over when de-referencing a soft pointer. The average skip distance of SKL is defined accordingly. Figure

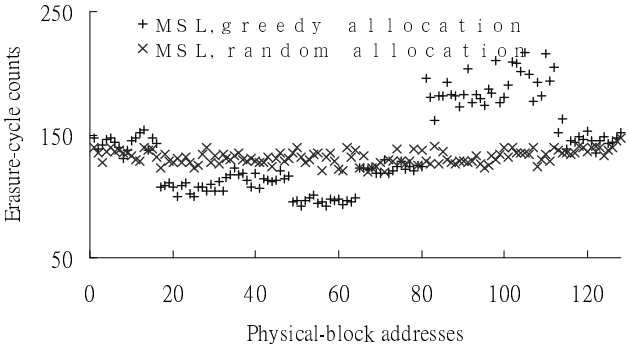

Figure 13: The distributions of erasure-cycle counts with random allocation (for MSL) and greedy allocation (for MSL). The access pattern is to update with a normal distribution.

11(b) shows the average skip distances with respect to different levels. The total number of levels is five. We can see that, even with the presence of short skips (as mentioned in 3.2.2), MSL still skips further than SKL at every level.

The next part of experiments are on update operations. Experimental setup is the same as that of SSL/LOL. Figure 12(a) shows the overheads of MSL and SKL with respect to different total numbers of levels. As Figure 12(a) shows, MSL still read much fewer words than SKL in all cases. Even with the extra cost of finding immediate predecessors for updates, the results in Figure 12(a) are close to that in Figure 11(a). It means that the overheads of finding predecessors are insignificant. Like the experiments for read-only queries, both MSL and SKL greatly reduce the read overheads as the total number of levels is large. MSL still outperforms SKL in terms of the average skip distances at every level, as shown in Figure 12(b). And, as expected, MSL writes slightly more words than SKL, but their erasure overheads are close.

Note that our results on MSL/SKL should not be interpreted as showing the benefits of using high-level SKL. After all, they are fundamentally different approaches. Not to mention that SKL requires RAM-resident translation tables and initialization scan.

4.4 Wear Leveling

This section is to verify whether the proposed MSL with random allocation evenly erase every NOR-flash blocks. In our approach, the key to achieve wear leveling is writing new objects to randomly allocated free space. If a block is chosen for space allocation but it does not have any, then the block becomes a victim for erasure. The policy is referred to as "random allocation". For comparison, we evaluated MSL with another policy that always allocate free space from one block until the block runs out of free space. On garbage collection, a block having the largest number of invalid objects is chosen for erasure. This policy is referred to as "greedy allocation". The access pattern is to update with a normal distribution.

Figure 13 compares the distribution of blocks' erasure-cycle counts of the two policies. As expected, our approach achieve a fairly even distribution of erasure-

cycle counts. On the other hand, greedy allocation results in very large variances among erasure-cycle counts. That is because, under normal distribution, some particular objects are frequently updated. As a result, with greedy allocation, invalid objects would possibly be clustered in some particular blocks. These blocks are preferred by garbage collection and thus are repeatedly erased.

5 Conclusions

To embedded devices, efficient data organization means not only reduced CPU cycles but also prolonged operating periods. Index structures for byte-addressable RAM or block-oriented storage are not applicable to NOR flash memory, because in-place updates are prohibited. On NOR flash, the problem is that data updates and pointer updates may recursively trigger each other. Past work tackle this problem by using logical addresses to detach the two from each other. The price paid to is an extra RAM-resident translation table and a lengthy scanning procedure on initialization. This work considers a native index structure for NOR flash. Our approach is based on physical addresses, and therefore the needs for logical addresses are completely removed. It is achieved by allowing a number of probes when de-referencing a data pointer. It prevents our index structure from being affected by garbage-collection activities. Surprisingly, its nature of randomness greatly benefits data-access performance. We have conducted a series of experiments and comparisons, for which we have very encouraging results.

References

[1] A. Kawaguchi, S. Nishioka, and H. Motoda,"A Flash-Memory Based File System," Proceedings of the USENIX Technical Conference, 1995.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms," The MIT Press, 1990.

[3] Samsung Electronics Company, "K8C1215ETM 32M x16 MLC NOR Flash Data Sheet".

[4] Samsung Electronics Company, "K4S561632J 4M x 16Bit x 4 Banks SDRAM Data Sheet".

[5] S. Lin, D. Zeinalipour-Yazti, V. Kalogeraki, D. Gunopulos, W. A. Najjar, "Efficient Indexing Data Structures for Flash-Based Sensor Devices," ACM Transactions on Storage, Volume 2 , Issue 4, 2006.

[6] C. H. Wu, T. W. Kuo, and L. P. Chang, "An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems," ACM Transactions on Embedded Computing Systems, Volume 6, Issue 3, 2007.

[7] X. Y. Xiang, L. H. Yue, Z. Z. Liu, and P. Wei, "A Reliable B-Tree Implementation over Flash Memory," in Proceedings of the ACM Symposium on Applied Computing, 2008.

[8] C. H. Wu, T. W. Kuo, and L. P. Chang "The Design of efficient initialization and crash recovery for log-based file systems over flash memory," ACM Transaction on Storage, Volume 2, Issue 4, 2006.

[9] K. S. Yim, J. H. Kim, and K. Koh, "A Fast Start-Up Technique for Flash Memory Based Computing Systems," in Proceedings of the ACM Symposium on Applied Computing, 2005.

[10] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," Communications of the ACM, Vol. 33, No. 6, 1990.

[11] L. P. Chang and T. W. Kuo, "Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation," ACM Transactions on Storage, Volume 1, Issue 4, 2005.

[12] Li-Pin Chang , "On Efficient Wear-Leveling for Large-Scale Flash-Memory Storage Systems," in Proceedings of the 22nd ACM Symposium on Applied Computing, 2007.

[13] D. W. Jung, Y. H. Chae, H. S. Jo, J. S. Kim, and J. W. Lee, "A Group-Based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems," in Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, 2007.

[14] Y. H. Chang, J. W. Hsieh, T. W. Kuo, "Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Leveling Design," in Proceedings of the 44th Annual Conference on Design Automation, 2007.

[15] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compactflash Systems," IEEE Transactions on Concumer Electronics, Vol. 48, No. 2, 2002.

[16] S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. W. Park, and H. J. Song, "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation," ACM Transactions on Embedded Computing Systems, Vol 6, Issue 3, 2007.

[17] E. Gal and S. Toledo, "A Transactional Flash File System for Microcontrollers," in Proceedings of the USENIX Annual Technical Conference, 2005.

[18] D. Woodhouse, "JFFS: The Journalling Flash File System," Proceedings of Ottawa Linux Symposium, 2001.